L17 - Error Handling II

March 12, 2020

1 Handling Errors II

1.1 Assertions

- used for internal consistency checks, such as the effect of a mutator method
- normally removed before code goes into production
  - some sort of compiler option

1.1.1 The `assert` statement

- two forms
  - `assert <BooleanExpression>`
  - `assert <BooleanExpression> : <message>`
- the `<BooleanExpression>` should be true at this point
- `<message>` will be passed to the constructor of `AssertionError` to make a diagnostic message
  - An `AssertionError` is thrown if `<BooleanExpression>` is false

[ ]:
```
/**
     * Remove an entry with the given key from the address book.
     * @param key One of the keys of the entry to be removed.
     * @throws AssertionError
     */
    public void removeDetails(String key)
    {
        if ( key == null )
        {
            throw new IllegalArgumentException("USEFUL MESSAGE");
        }

        if (keyInUse(key))
        {
            ContactDetails details = book.get(key);
            book.remove(details.getName());
            book.remove(details.getPhone());
            numberOfEntries--;
```

```
        }
        // assertions

        assert !keyInUse(key); // better not be now

        // check that the book is a reasonable size
        // consistentSize is some helper function
        assert consistentSize(); : "Inconsistent book size"
    }
```

1.1.2 Assertion guidelines

- *not* an alternative to exceptions
- Use for internal checks. Remove from production code.
- Don't implement actual functionality in assertions.

1.2 Error Recovery

Client objects and programs should take note of the error notifications they receive, by checking return values. It is not generally good practice to ignore exceptions. You can include some code to attempt a recovery from the exception (probably some loop).

[ ]:
```
// try to save the addressbook

boolean success = false;
int attempts = 0;

do
{
    try
    {
        contacts.saveToFile(filename);
        successful = true;
    }
    catch( IOException e )
    {
        // a recovery routine
        System.out.println("Unable to save to " + filename);
        attempts++;
        if (attempts < MAX_ATTEMPTS)
        {
            filename = getNewFilenameSomehow();
        }
    } while (!successful && attempts < MAX_ATTEMPTS);

    if (!successful)
```

```
    {
        reportLackOfSuccessWompWomp();
        giveUp();
    }
}
```

1.3 Error Avoidance

Clients mught be able to use queries on the server to check data.

- a more robust client is more trustworthy from the server
- unchecked exceptions can be used in the client
- simpler client logic

Note that this might mean there is a higher degree of coupling between the client and server, and more methods that could have bugs.

1.4 File IO

Reading and writing to and from files breaks all the time because you're depending on the file existing, file permissions, not already open, etc.

The `Java.io` package supports IO and has the checked exception `java.io.IOException`:

- `java.io.File` provides information about files, folder, and directories
  - alternative: `java.nio.file.Path`
- `File` is a class, `Path` is an interface

1.5 Readers, writers and streams

- *readers* and *writers* are classes dealing with IO from text files, based around a file full of `char` type things.
- *streams* deal with binary data, based around the `byte`

1.6 Writing to files

The `FileWriter` object deals with There are three stages to writing to a file:

- open the file (`new FileWriter(some_filename)`)
- write to the file (`FileWriter.write("Something to write")` for a text file)
- close the file (`FileWriter.close()`)

1.6.1 Text output to file - try-with

[3]:
```
Boolean write_the_thing = true;

try (FileWriter writer = new FileWriter("file.txt"))
{
    while(write_the_thing)
    {
        writer.write("m");
        write_the_thing = false;
    }
}
```

1.7 Text inputs

Use `BufferedReader` objects for line-based inputs. Again: open, read, close. Failure throws an `IOException`.

You will need to pick a `charset` to read. This establishes how text is encoded with binary.

[ ]:
```
// basic file reading pattern

Charset charset = Charset.forName("US-ASCII");
Path path = Paths.get(filename);

BufferedReader reader = Files.newBufferedReader(path,charset);
String line = reader.readLine();

while( line != null )
{
    do_something(line);
    line = reader.readLine();
}

reader.close();
```

1.8 Text input from the terminal

`System.in` maps to the terminal (similar to `System.out` for writing stuff) - it has the type `java.io.InputStream`.

We can use `Scanner` to parse text input - it has methods like `nextInt()` and `nextLine()` that hopefully do the obvious thing.

[4]:
```
Scanner reader = new Scanner(System.in);
```

[9]:
```
String words = reader.nextLine(); the next complete line, not including a␣
↪newline
```

```
What's up, doc?
```

[10]:
```
System.out.println(words);
```

```
What's up, doc?
```

[ ]:
```

```